Assignment 3

Software Development 2023

Department of Computer Science
University of Copenhagen

Mads-Ulrik Rasmussen <mura@di.ku.dk>
Cecilie Hundahl Johnsen <cejo@di.ku.dk>

Version 1.0: February 17th
Due: February 24th, 2023 kl 15.00

Contents

Please familiarize yourself with the assignment as a whole, before you start working on your solution.

1 Hand-out and Git Classroom

The assignment template can be found by selecting your KU-id at `https://classroom.github.com/a/y3Az9Dcn`. The resulting repository is where you will work, test, and hand in your final code (See the Submission section for details and specific requirements).

When you push to your repository an Autograder will attempt to test-run your program and tests, as well as run some tests defined by your TA's. All tests should be successful before final turn-in (i.e. there should be shown a small checkmark next to your latest commit).

It is important that your methods and constructors have the exact signatures handed out and described below, or the tests will fail.

2 Sorting cards

The first task concerns the implementation of interfaces. Interfaces provide a way of specifying a *contract* that the implementing object has to satisfy. The code for the first task can be found in the `SortingCards` folder from the hand-out. The folder contains the files `Card.cs` and `Program.cs`.

2.1 Card.cs

This file contains a partial implementation of the `Card` class, which represents an abstraction of a French-suited playing card. `Card` contains two public fields:

- `Card.rank`: An integer in the interval $[1, 13]$, representing the rank of the card. Ace has a rank of 1, and face cards have a rank of 11, 12 and 13.
- `Card.suit`: An integer in the interval $[1, 4]$, representing the suit of the card. The suits are enumerated in the order Spades = 1, Hearts = 2, Clubs = 3, Diamonds = 4.

It is not necessary to consider the handling of values that lie outside the specified intervals.

Furthermore, `Card` implements the interface `IComparable`[1], which defines a comparison method `CompareTo` that can be used to compare two objects of the same type. In a way, this interface can be considered a *contract*, which specifies a set of conditions (i.e., the existence and behavior of a certain method) that the implementing object should satisfy.

All objects that implement the `IComparable` interface can be sorted using the `List.Sort` method, so any two cards can be sorted (compared) in this manner:

2.1. If their suits are different, they are sorted based on the enumeration of their suits (e.g., Spades are less than Diamonds).

[1] https://learn.microsoft.com/en-us/dotnet/api/system.icomparable?view=net-6.0

2.2. If their suits are identical, they are sorted based on their rank, i.e., lower ranks are less than higher ranks.

The current implementation of `Card.CompareTo` is a stub. **Your task** is to finish the implementation of `CompareTo`, such that it implements the desired behavior as specified in the `IComparable.CompareTo` documentation[2].

2.2 Program.cs

The `Main` function in `Program.cs` contains an example usage of `List.Sort`. which can be used to verify the correctness of your implementation. You are *not* expected to write your own tests for *this* task, and this task is not included in the report.

3 TicTacToe

We will now introduce you to an incomplete version of the game Tic Tac Toe. In this assignment, you will go through testing and implementing the missing functionality in the game. This is the first time we introduce you to a slightly bigger system contra the small classes and methods you have created so far.

This assignment will focus on a test-driven approach, where you will write tests before implementing functionality, so remember to read the **whole** assignment before working on your solution. Specifically, remember to create at least one test for each **Move()**-method before implementing the functionality.

3.1 Cursor

In this part, you will be extending the code base with four tests and the associated five methods they test, which correspond to the 4 directions that the cursor can move, as well as a method that combines these 4 functionalities. To understand how the cursor moves, open the `Cursor.cs`-file (in `/IO/`), where all the implementation of this part of the assignment resides. Initially, the position of the cursor is (0, 0), corresponding to the top left corner of the board. The position consists of an X- and Y-coordinate and to navigate around the board, these values should be incremented or decremented corresponding to which direction the cursor should move.

However, **do not implement the Move-methods yet!** Start by going to the file `CursorTest.cs`. In the `SetUp`, a Cursor has been instantiated and starts out in the center of the board with the position (1,1). In each of the `Move_()`-tests, call the corresponding `Move_()`-function from `Cursor.cs` on the Cursor from SetUp, and make a boolean comparison to assert that the position of the Cursor after the call to `Move_()` has changed correctly. Follow the Arrange, Act, Assert annotation of your test, from Chapter 5 in Adaptive Code). This could be an example:

[2]https://learn.microsoft.com/en-us/dotnet/api/system.icomparable?view=net-6.0

```
[Test]
public void MoveUpTest() {
    // Move the cursor up.
    // Save the position of the Cursor in a variable.
    // Move the cursor up again.
    // Assert.True() to compare variable and position of cursor.
}
```

To run the tests use `dotnet test` inside the solution folder or test project folder.

Initially, your tests will **not** pass, since the actual `Move_()`-method has not been implemented yet - this is Test-Driven Development! Also, in the example above, the cursor will move out of the board if the `MoveUp()`-method hasn't been implemented correctly, and you should design your tests to allow you to catch incorrect implementations. Then, when implementing functions, you can continuously make sure that the implementation works as intended.

When you have written your tests, it is time to implement each of the 4 `Move()`-functions in `Cursor.cs`, so that each cell can be reached, but make sure that the cursor cannot leave the board. Since we are doing Test-Driven Development, add some code, then run your tests and see if they pass.

Next, implement the `MoveCursor(InputType inputType)`-function by using a control flow (either a `If-Else`-statements or a `Switch`-statement), by using the Move()-functions you just implemented. Each branch should compare the argument `InputType` with each corresponding type of the `InputType` enum.

Hint: Note the return type of the function! The `InputType.PerformMove` case should return `False` and all other cases should return `True`.

3.2 BoardChecker

For this part of the assignment, you will be implementing four functions in `BoardChecker.cs`. Three of the functions are being made to check the three winning conditions of the game: Vertically, Horizontally and Diagonally. For this part, you do not need to do Test-Driven Development, but consider if it will help you implement this functionality more efficiently (**Spoiler Alert:** It will)

Hint: Consider using the `Get()`-method from the `Board.cs` class.

Now it is time to implement the tests that show your BoardChecker works. Open the `BoardCheckerTest.cs`-file, where the unimplemented tests reside. Finish the five tests for diagonal, row, and column wins as well as a test for the tie and inconclusive outcomes. The way you will be testing this is by setting up a board position using the `TryInsert()`-method from the `Board.cs` class.

You do not have to make exhaustive tests that show it will work for every board position. You have to test for some some of these, to cover different sce-

narios.

When testing you should use `Assert.AreEqual` to test if what the `CheckBoardState` method returns is the correct `BoardState`.

3.3 Bonus

Create even more test cases! Which kinds can you come up with? Which outcomes of the functions that you implemented has not been covered?

3.4 The Main Method

Nothing has to be implemented inside the main method, it will just be helpful to look at. Since it describes the controls for the game via the KeyToMoveMap class and describes which players participate in the game.

4 Deliverables

- Sorting cards:
  - Implement Card.CompareTo
- TicTacToe:
  - Implement the four missing tests in `CursorTest.cs`.
  - Implement the five missing methods in `Cursor.cs`.
  - Implement the five missing tests in `BoardCheckerTest.cs`.
  - Implement the four missing methods in `BoardChecker.cs`.
  - Write the 2-3 pages tech report as described below.
- Successful Git Classroom tests.

5 Writing a Report

Being able to construct a well-written report documenting your work is a crucial aspect of sustainable software development. You will be writing a report for the exam and if you are interested in studying Computer Science, you will likely write technical reports in other courses. Furthermore, your report should follow the report formatting specified in course guidelines[3].

To help you get started, we have supplied the `techReport.pdf` which will give you an overview of how to structure your technical reports within the scope of this course. You should look through this before reading onward.

[3] `https://github.com/diku-dk/su-guides/blob/main/guides/ReportFormatting.md`

Note that this report should only concern your work on the **TicTacToe** task, and not the **Sorting cards** task.

Your report should follow the structure specified in `techReport.pdf`. For **this** assignment, your report should roughly look as follows:

- An **Introduction**, where you introduce the "problems" you aim to solve (*i.e. implementing missing parts of TicTacToe and test it thoroughly*) and a glimpse at the extent of your solution.
- **Implementation**, in-code examples of how you chose to solve the presented problem, any code you might want to explain or showcase, any issues you may have faced and solved with code.
- **Evaluation**, this is the section that includes a description of your tests and their result. Furthermore, you have to disambiguate any ambiguities, evaluate on your work, if there is anything you might not have been certain of, something you interpreted in a different way, or perhaps any alternate solutions.
- **Conclusion** Briefly sum up the report, your results and the work you have done.

Note that Background, Analysis, and Design are not included. Your report should be 2-3 pages and *must* be submitted as a *pdf*.

6 Submission

Before you submit anything, make sure that you have followed the official style guide. [4] Read your work thoroughly, edit spelling mistakes, and set it up in a way that makes it possible for another computer scientist to open your solution and understand it easily.

Your work must be submitted through both Absalon and Github Classroom. Git Classroom should show successful tests and contain:

- Your executable program.
- A README containing instructions on how to run your program and other relevant information you feel we should know.
- Any other files which were included in the hand-out template.

On Absalon you should submit three files:

- Your report as a pdf called `yourName-A3.pdf`. The first letter needs to be lowercase.
- Your repository as a zip-file called `yourName-A3.zip`.

[4] `https://github.com/diku-dk/su-guides/blob/main/guides/CSharpStyle.md`

- A txt file containing your name, ku-id and a link to your assignment on GitHub. This should be handed in **separately** (not in the zip file).

When submitting code make sure you only submit what is required to run the code. That is usually the `.csproj` as well as any `.cs` files. Exclude any OS specific files. You can run the command `dotnet clean` from the directory containing your code. When in doubt, attempt to simulate running the code from the zip file, i.e., copy, extract, and run.